SPRING-BOOT TUTORIAL

CORE JAVA VS SPRING

Core Java is powerful but requires a lot of manual work for enterprise-level applications. Spring Framework simplifies this by offering structured solutions for common problems.

Main Advantages of Spring Over Core Java:

Dependency Injection (DI):

- Core Java: You manually create and manage object dependencies.
- Spring: Automatically injects dependencies using annotations or XML, reducing boilerplate code.

Modular Architecture:

- Spring provides modules like Spring MVC, Spring JDBC, Spring Security, etc., making it easier to build scalable applications.

Transaction Management:

- Core Java: You handle transactions manually.
- Spring: Offers declarative transaction management using annotations.

Integration Support:

- Spring integrates easily with databases, messaging systems, and other frameworks.

Testability:

- Spring applications are easier to test due to loose coupling and support for mock objects.

SPRING VS SPRINGBOOT

Spring Boot builds on top of Spring Framework and removes the complexity of configuration and setup, making development faster and easier.

Main Advantages of Spring Boot Over Spring:

Auto Configuration:

- Spring Boot automatically configures your application based on the dependencies you add, saving time and effort.

Embedded Servers:

- Comes with built-in servers like Tomcat, Jetty, or Undertow.
- No need to deploy WAR files to external servers—just run the app as a Java program.

Starter Dependencies:

- Spring Boot provides "starter" packages (e.g., spring-boot-starter-web, spring-boot-starter-data-jpa) that bundle common dependencies for specific tasks.

Production-Ready Features:

- Includes metrics, health checks, and externalized configuration support out of the box.

Simplified Deployment:

- You can package your application as a JAR and run it directly, making deployment easier.

Microservices Friendly:

- Spring Boot is ideal for building microservices due to its lightweight nature and fast startup time.

SUMMARY:

FEATURE	CORE JAVA	SPRING FRAMEWORK	SPRING BOOT
Configuration	Manual	Semi-automated	Fully automated
Dependency Management	Manual	Supported	Simplified with starters
Server Setup	External	External	Embedded
Microservices Support	Limited	Possible	Built-in
Development Speed	Slow	Moderate	Fast
Learning Curve	Steep	Moderate	Easy

SPRING BOOT ARCHITECTURE:

1. Client Layer

- This represents the external system or user that interacts with the application by sending HTTPS requests.

2. Controller Layer (Presentation Layer)

- Handles incoming HTTP requests from the client.
- Processes the request and sends a response.
- Delegates business logic processing to the Service Layer.

3. Service Layer (Business Logic Layer)

- Contains business logic and service classes.
- Communicates with the Repository Layer to fetch or update data.
- Uses Dependency Injection to get required repository services.

4. Repository Layer (Data Access Layer)

- Handles CRUD (Create, Read, Update, Delete) operations on the database.
- Extends Spring Data JPA or other persistence mechanisms.

5. Model Layer (Entity Layer)

- Represents database entities and domain models.
- Maps to tables in the database using JPA/Spring Data.

6. Database Layer

- The actual database that stores application data.
- Spring Boot interacts with it through JPA/Spring Data.

REQUEST FLOW IN SPRING BOOT:

CLIENT → CONTROLLER → SERVICE → REPOSITORY → DATABASE → RESPONSE

- A user sends a request (e.g., to get user details).
- The **Controller** receives it and calls the **Service**.
- The **Service** processes logic and asks the **Repository** for data.
- The **Repository** fetches data from the **Database** using the **Model**.
- The result is sent back through the layers to the **Client**.

INVERSION OF CONTROL:

- In Spring Framework, the **IoC (Inversion of Control) Container** is the heart of the system.
- It's responsible for creating and managing objects, known as **beans**, and injecting their dependencies automatically.
- Instead of developers manually creating and connecting objects, the **container** takes control—hence the term ***Inversion of Control*.**
- This is done using **Dependency Injection (DI)**, which can be configured through XML files, Java-based configuration, annotations, or simple POJOs.

TYPES IOC CONTAINERS

1. BeanFactory:

- The BeanFactory is the most **basic version of the IoC container**.
- It provides basic support for **dependency injection and bean lifecycle management**.
- It is suitable for **lightweight applications** where advanced features are not required.

2. ApplicationContext:

- The ApplicationContext is an **extension of BeanFactor**y.
- More enterprise features like **event propagation, Event handling, internationalization** and more.
- It is the preferred choice for most Spring applications due to its **advanced capabilities**.

KEY FEATURES OF IOC CONTAINER

Dependency Injection

- Automatically connects required objects.

Lifecycle Management

- Handles bean creation, initialization, and destruction.

Flexible Configuration

- Supports both **XML** and **annotations**.

Loose Coupling

- Objects are independent and easily replaceable.

DEPENDENCY INJECTION:

Dependency Injection (DI) is a design pattern in which objects receive their dependencies from an external source rather than creating them internally. It promotes loose coupling, easier testing, and better code maintainability. In Spring, DI is achieved mainly through Constructor Injection or Setter Injection.

NEED FOR DEPENDENCY INJECTION:

- In real-world applications, classes often depend on other classes to perform tasks. For example, **Class One** may need **Class Two** to work properly.
- If Class One creates Class Two directly, it leads to **tight coupling**—making the code harder to change, test, or maintain.
- **Tight coupling can cause:**
    - Difficulty in updating code
    - Reduced testability
    - Higher chances of system failure
- To avoid this, **direct dependencies should be avoided**.
- **Spring IoC (Inversion of Control)** solves this by using **Dependency Injection (DI)**:
    - The container creates and provides the required objects.
    - The developer just defines what is needed.
- DI helps in:
    - **Loose coupling** between classes
    - **Easier testing** and mocking
    - **Better code reuse and flexibility**
- Loose coupling is achieved by:
    - Using **interfaces** for common tasks
    - Letting **Spring container** inject the correct implementation

TYPES OF DEPENDENCY INJECTION:

1.CONSTRUCTOR INJECTION:

- In this method, dependencies are passed to the class through its constructor.
- This ensures that all required dependencies are provided when the object is created.
- You can declare the dependency fields as final, which ensures they **cannot be reassigned** after the object is created.
- As a result, the class becomes **immutable**—its state doesn't change after construction, which is a good practice in software design.
- Since all dependencies are passed through the constructor, you can easily **provide mock objects** during unit testing. You don't need to rely on the Spring container or annotations like @Autowired to inject dependencies.
- Constructor Injection also makes it **clear what dependencies** the class needs, improving readability and maintainability.

2.SETTER INJECTION:

- Here, dependencies are provided using setter methods after the object is created.
- This is useful when some dependencies are optional or can be changed later. However, it's less safe because the object can exist without having all its dependencies set.
- **Good for Optional Dependencies**: You can choose which dependencies to set, making it flexible.
- **Risk of Incomplete Objects**: If a required dependency is not set, the object may not work properly.
- **Readable Code**: Setter methods clearly show what dependencies are being injected.
- **Less Safe**: No guarantee that all dependencies are provided before use.
- **Allows Reconfiguration**: You can change dependencies later if needed.
- **Slightly Harder to Test**: You need to manually set dependencies in tests, which can be error prone.

3. FIELD INJECTION:

- Dependencies are injected directly into class fields using annotations like @Autowired.
- While it's quick and requires less code, it's not recommended.
- **Breaks Encapsulation**: Fields are accessed directly, which goes against object-oriented principles.
- **Harder to Test**: You can't easily pass mock dependencies during unit testing.
- **Hidden Dependencies**: It's not clear what the class depends on just by looking at the constructor or methods.
- **Less Flexible**: You rely more on the Spring container, making the class harder to use outside of Spring.

BEAN LIFE CYCLE IN JAVA SPRING:

The bean lifecycle in Spring is the sequence of steps a bean goes through from creation to destruction, and it's managed by the Spring container.

Bean Life Cycle Phases:

1. **Container Started**: Spring IoC container is initialized and ready to manage beans.
2. **Bean Instantiated**: Spring creates an object (bean) using the class definition.
3. **Dependencies Injected**: Required dependencies are injected into the bean (via constructor, setter, or field).
4. **Custom Initialization**: If defined, Spring calls:
    1. init () method
    2. @PostConstruct annotated method
    3. afterPropertiesSet() from InitializingBean interface
5. **Bean is Ready**: Bean is fully initialized and ready to be used in the application.
6. **Custom Utility Methods**: You can call any custom methods defined in the bean during its usage.

7. **Custom Destruction**: When the container shuts down, Spring calls:
    1. destroy() method
    2. @PreDestroy annotated method
    3. destroy () from DisposableBean interface

BEAN LIFE CYCLE IMPLEMENTATION:

Spring allows you to define custom init() and destroy() methods for beans. These methods run when the bean is created and destroyed. You can implement them in **three ways**:

USING XML CONFIGURATION:

- **How it works**: Define init-method and destroy-method in the Spring XML file.
- **Steps**:
    - Create a bean class with **init ()** and **destroy ()** methods.
    - Register these methods in the XML using:

```
<bean id="hw" class="beans.HelloWorld" init-method="init" destroy-method="destroy"/>
```

    - Load the XML in a driver class and call **close ()** to trigger destroy.
- **Pros**: Simple and clear for XML-based projects.
- **Cons**: Requires manual XML configuration.

USING INTERFACES (PROGRAMMATIC APPROACH):

- **How it works**: Implement InitializingBean and DisposableBean interfaces.
- **Steps**:
    - Override afterPropertiesSet() for init and destroy() for cleanup.
    - Define the bean in XML without specifying methods.
    - Call close() in the driver class to trigger destroy.
- **Pros**: No need to register methods in XML.
- **Cons**: Tightly couples your bean to Spring interfaces.

USING ANNOTATIONS:

- **How it works**: Use **@PostConstruct** for init and **@PreDestroy** for destroy.
- **Steps**:
    - Annotate methods in the bean class.
    - Add CommonAnnotationBeanPostProcessor in XML to activate annotations.
    - Call close() in the driver class to trigger destroy.
- **Pros**: Clean and annotation-based; no need for interfaces or method names in XML.
- **Cons**: Requires annotation support and extra bean processor.

SINGLETON AND PROTOTYPE BEAN SCOPES IN JAVA SPRING:

Bean Scope defines how many instances of a bean are created and how long they live. It's managed by the **Spring container** and controls the lifecycle and visibility of beans.

TYPES OF BEAN SCOPES:

SINGLETON (DEFAULT SCOPE):

- Only **one instance** of the bean is created per Spring IoC container.
- Every time the bean is requested, **the same instance is returned**.
- Spring checks if the bean already exists:
    - If yes → returns the existing instance.
    - If no → creates a new one (only once).
- Best for **stateless beans** (no internal state changes).
- **Default scope** in Spring.
- **Example Behaviour**: You call `getBean()` multiple times → same object returned each time.

PROTOTYPE:

- A **new instance** of the bean is created **every time** it is requested.
- Useful for **stateful beans** (each instance holds different data).
- Can be requested via:
    - getBean() method
    - XML-based dependency injection
- **Example Behaviour**: You call getBean() multiple times → different objects returned each time.

REQUEST (WEB-ONLY):

- A new bean instance is created for **each HTTP request**.
- Valid only in **web-aware Spring ApplicationContext**.
- A new bean instance is created for **each HTTP request**. It's perfect for handling **request-specific data**, like form submissions.

SESSION (WEB-ONLY):

- One bean instance per **HTTP session**.
- Lives as long as the user session is active.
- It's useful for storing **user-specific data** across multiple requests.

GLOBAL-SESSION (WEB-ONLY, MOSTLY FOR PORTLETS):

- One bean instance per **global HTTP session**.
- Used in **portlet-based applications**.

SPRING BEANS:

- **Spring Bean**: An object managed by the **Spring IoC (Inversion of Control) container**.
- Beans are created, assembled, and managed by Spring.
- Purpose: Simplifies dependency management and promotes loose coupling.

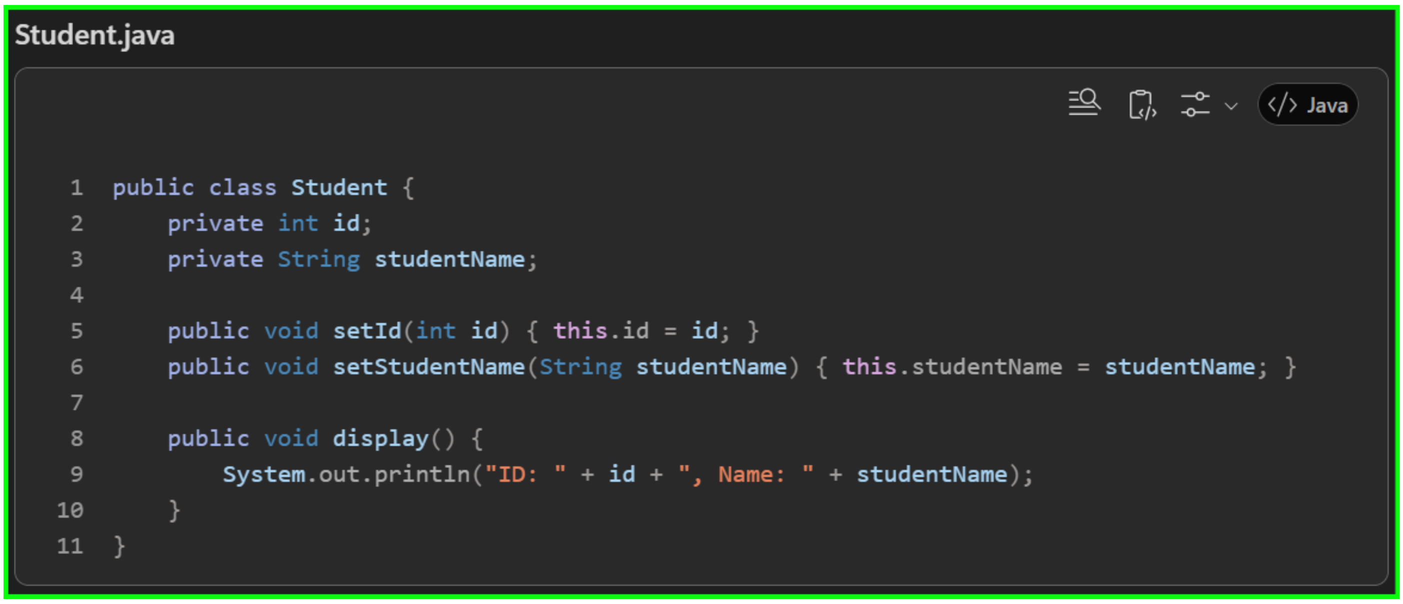
Methods to Create Spring Beans:

1. XML CONFIGURATION (BEANS.XML):

- Traditional way of defining beans.
- Beans are declared in an XML file using <bean> tag.
- Requires specifying id and class.

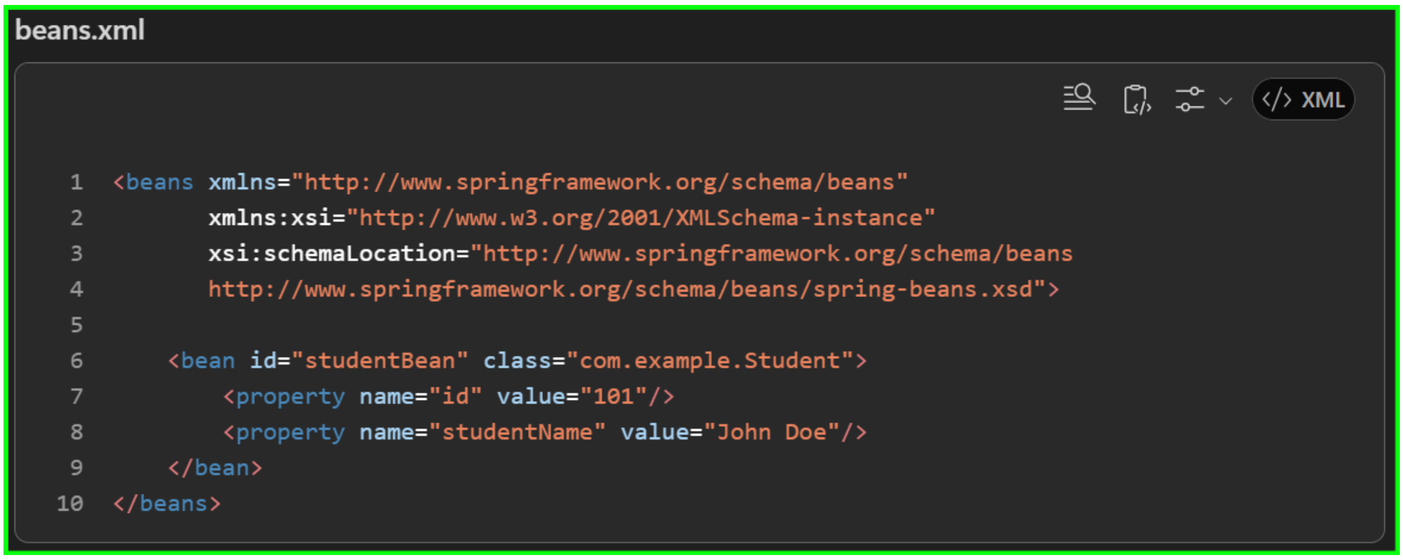
Steps:

- Create a POJO class.
- Define bean in beans.xml.
- Load context and get bean.



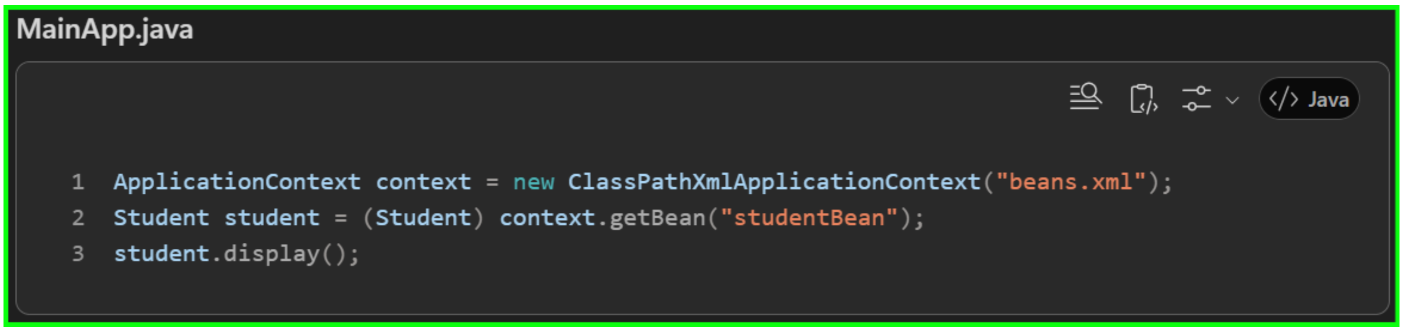
Student.java

```java
public class Student {
    private int id;
    private String studentName;

    public void setId(int id) { this.id = id; }
    public void setStudentName(String studentName) { this.studentName = studentName; }

    public void display() {
        System.out.println("ID: " + id + ", Name: " + studentName);
    }
}
```



beans.xml

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="studentBean" class="com.example.Student">
        <property name="id" value="101"/>
        <property name="studentName" value="John Doe"/>
    </bean>
</beans>
```


MainApp.java

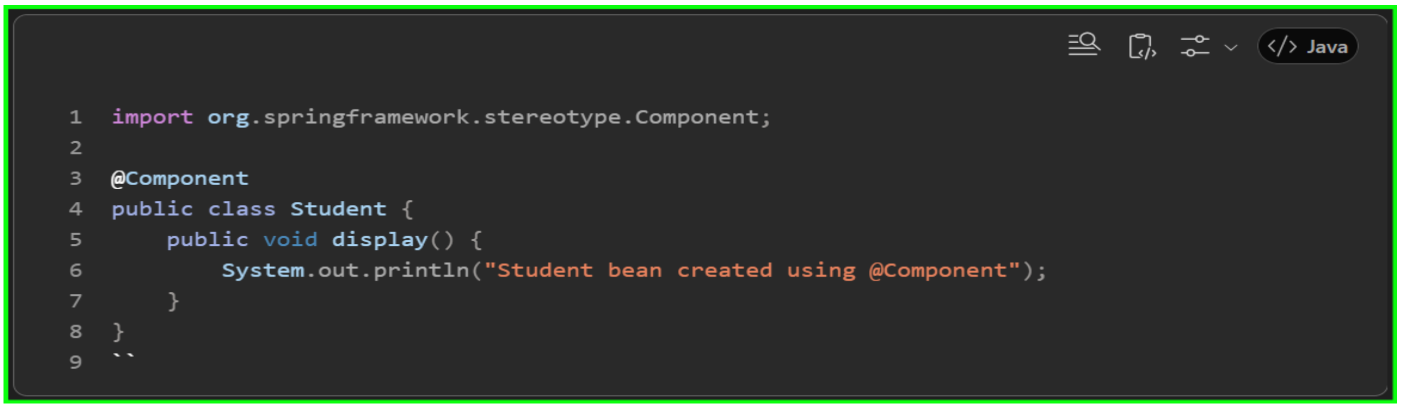
```java
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
Student student = (Student) context.getBean("studentBean");
student.display();
```

2. USING @COMPONENT ANNOTATION:

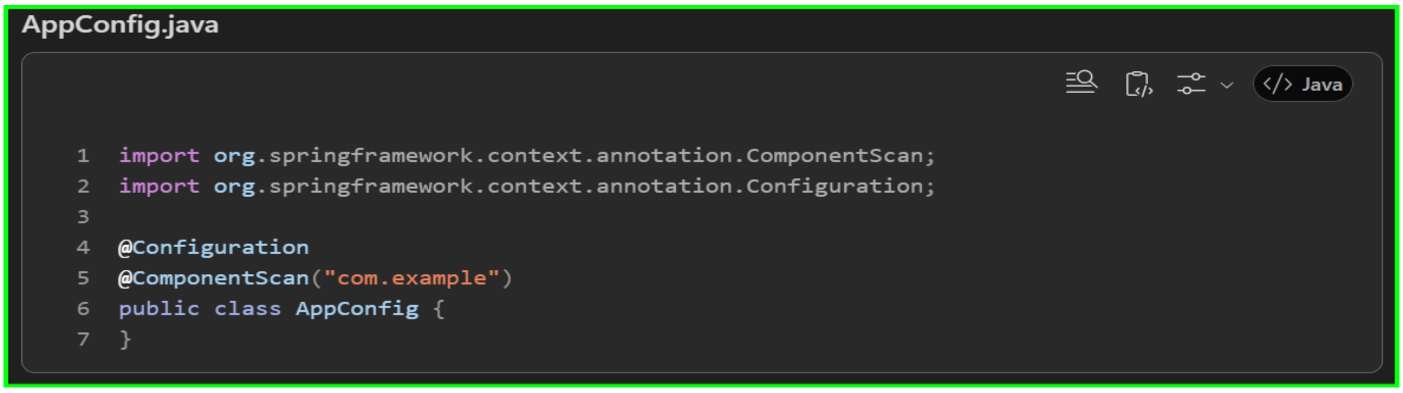
- Marks a class as a Spring-managed component.
- Requires @ComponentScan to detect beans automatically.

Steps:

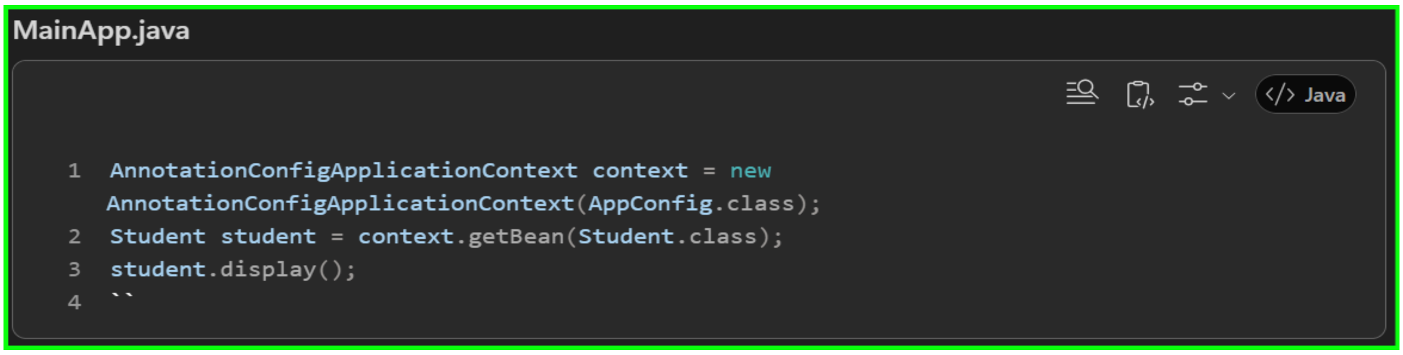
- Annotate class with @Component.
- EnableStudent.java**



```java
import org.springframework.stereotype.Component;

@Component
public class Student {
    public void display() {
        System.out.println("Student bean created using @Component");
    }
}
``
```


AppConfig.java

```java
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.example")
public class AppConfig {
}
```


MainApp.java

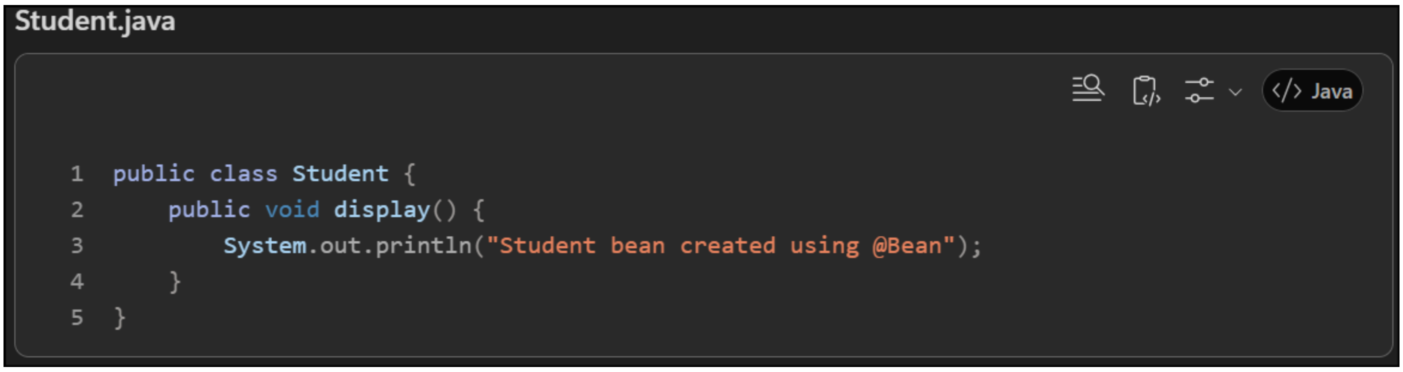
```java
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
Student student = context.getBean(Student.class);
student.display();
``
```

3. USING @BEAN ANNOTATION:

- Used inside a @Configuration class.
- Provides explicit control over bean creation.

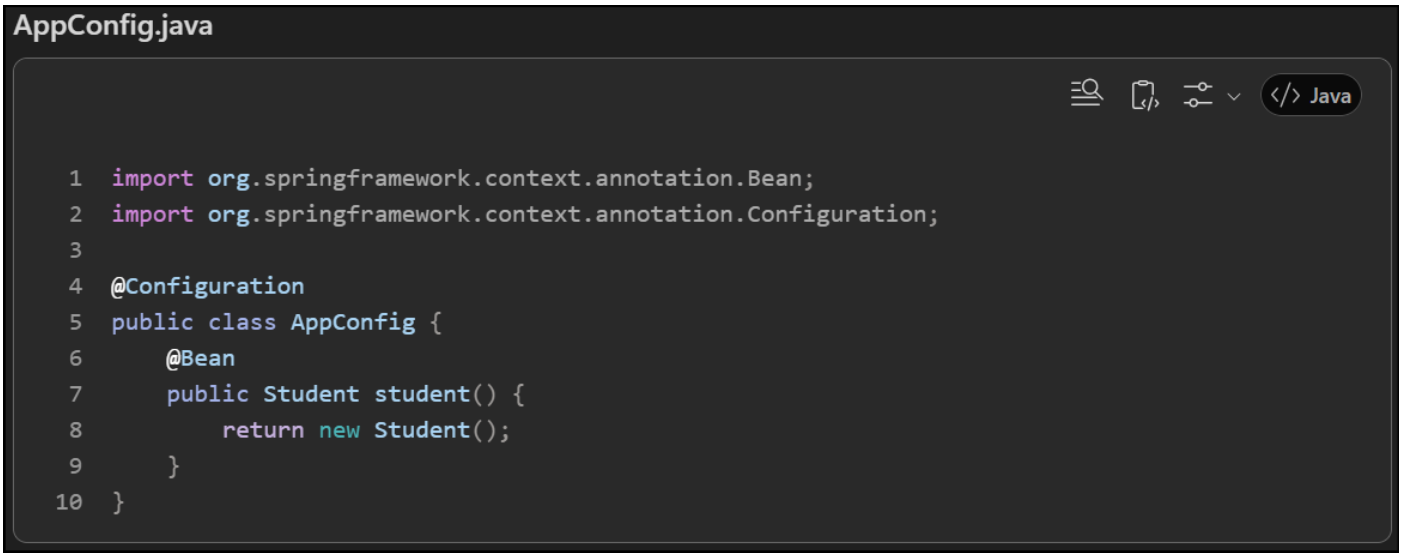
Steps:

- Create a @Configuration class.
- Define a method annotated with @Bean.



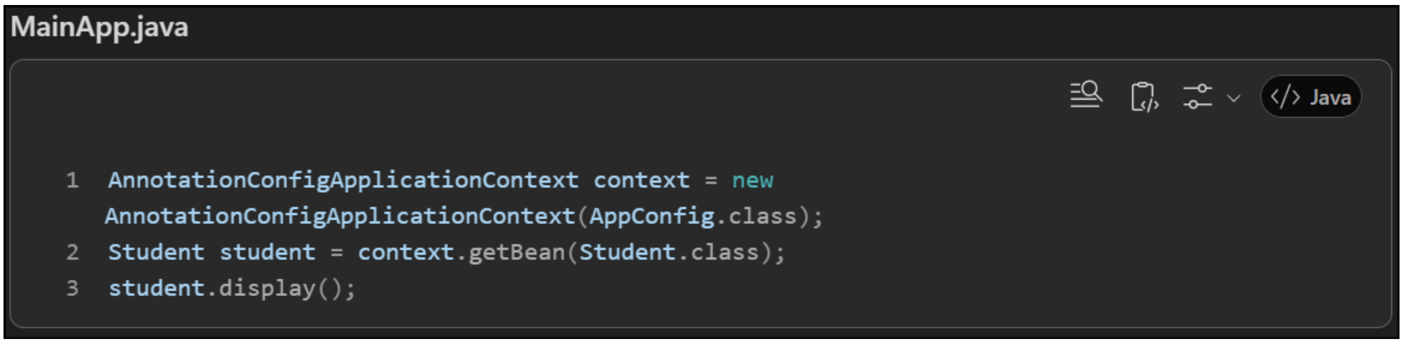
Student.java

```java
public class Student {
    public void display() {
        System.out.println("Student bean created using @Bean");
    }
}
```



AppConfig.java

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public Student student() {
        return new Student();
    }
}
```



MainApp.java

```java
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
Student student = context.getBean(Student.class);
student.display();
```

Key Differences:

- **XML: Legacy approach, still used in older projects.**
- **@Component: Automatic bean detection.**
- **@Bean: Explicit bean creation in Java config.**

AUTOWIRING IN SPRING FRAMEWORK:

- Definition: Autowiring is a feature in Spring that automatically injects dependencies into beans without explicit configuration.
- Purpose: Reduces boilerplate code by avoiding manual <property> or <constructor-arg> tags.
- How: Spring container detects dependencies and injects them based on type, name, or constructor.

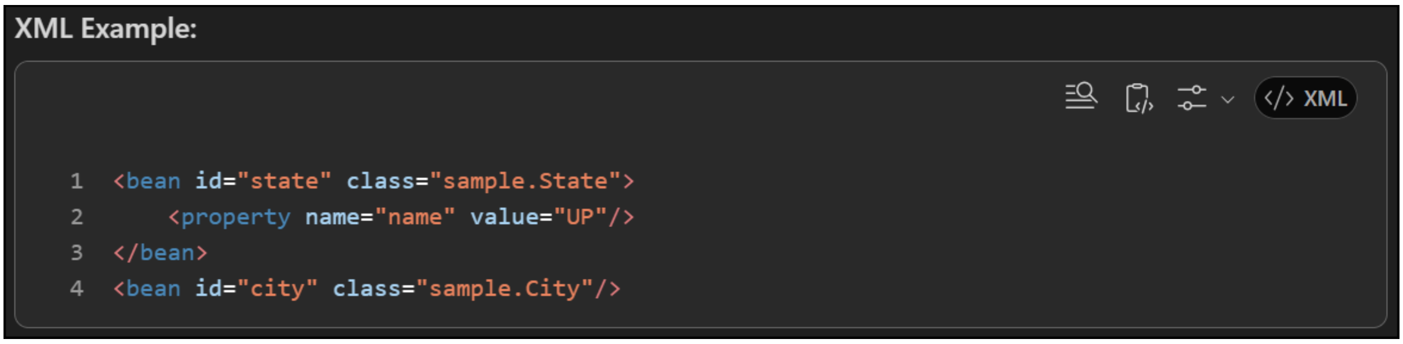
KEY CHARACTERISTICS:

- **ENABLED USING:**
  - **XML**: autowire attribute in <bean> tag.
  - **Annotation**: @Autowired on fields, constructors, or setters.
- Internally uses **constructor injection** when applied on constructors.
- **ADVANTAGES:**
  - Less code, easier configuration.
- **DISADVANTAGES:**
  - Less flexibility, harder to debug.
  - Cannot inject primitives or Strings automatically.

AUTOWIRING MODES:

1.NO (DEFAULT MODE):

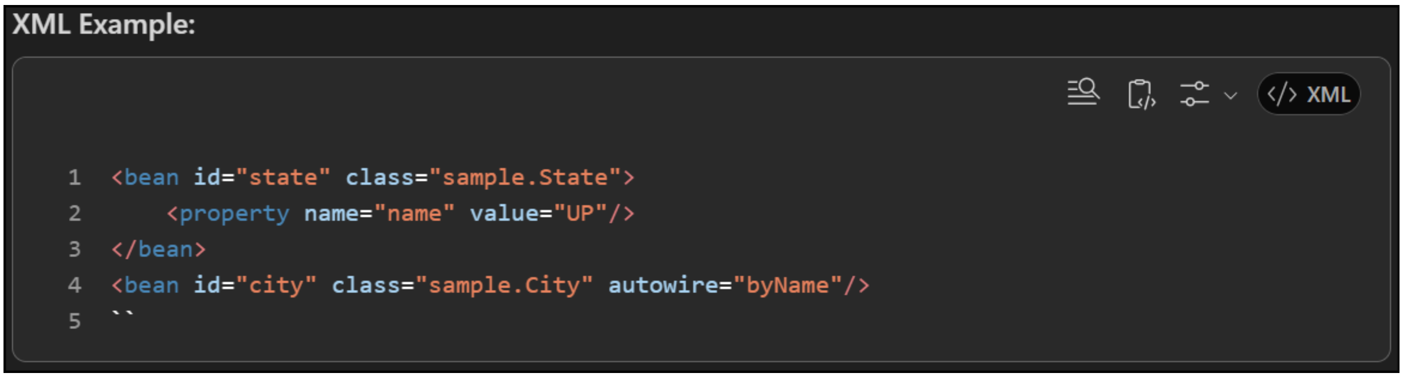
- No autowiring; dependencies must be set manually.
- It is the default mode used by Spring.


XML Example:

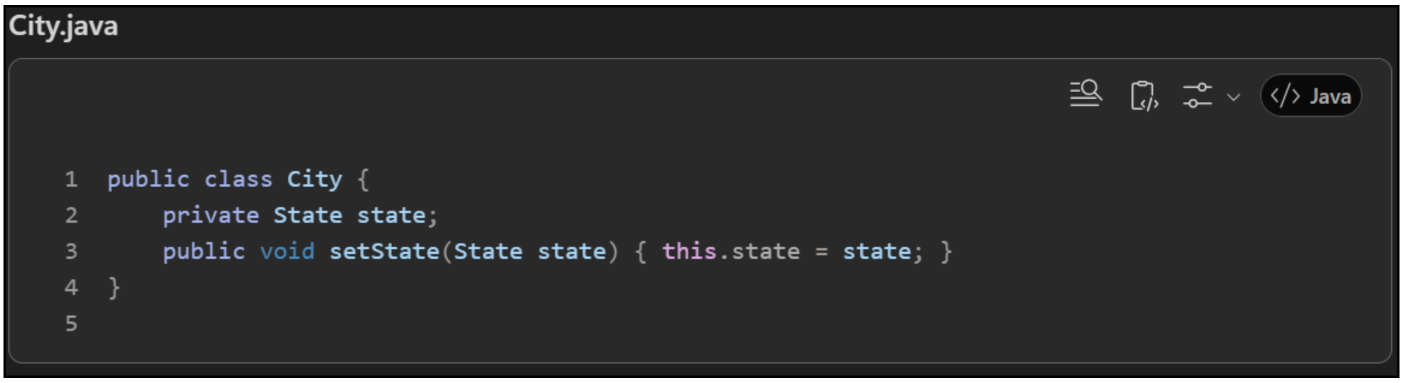
```xml
<bean id="state" class="sample.State">
    <property name="name" value="UP"/>
</bean>
<bean id="city" class="sample.City"/>
```

2. BYNAME:

- Matches bean name with property name.
- However, it requires that the name of the property and bean must be the same.
- It invokes the setter method internally for autowiring.


XML Example:

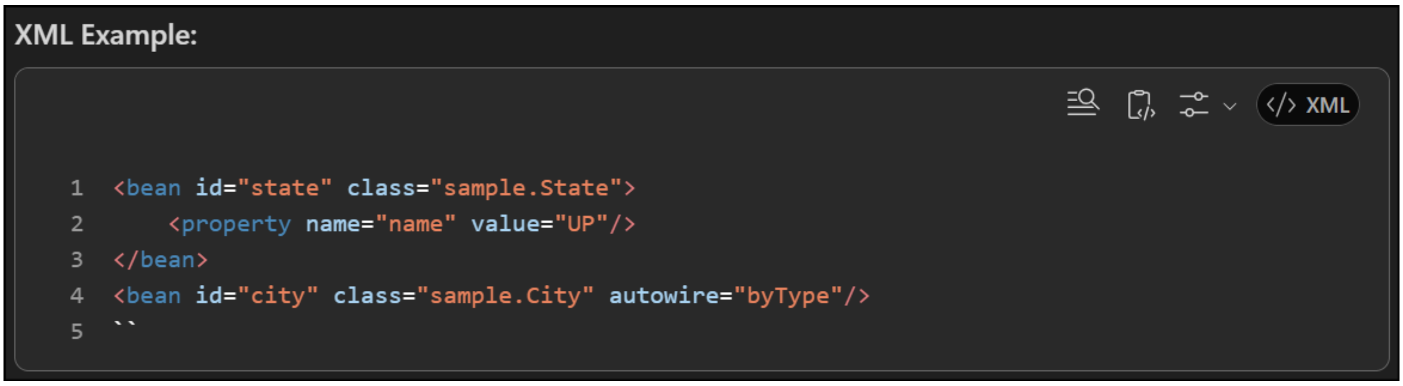
```xml
<bean id="state" class="sample.State">
    <property name="name" value="UP"/>
</bean>
<bean id="city" class="sample.City" autowire="byName"/>
``
```


City.java

```java
public class City {
    private State state;
    public void setState(State state) { this.state = state; }
}

```

3. BYTYPE:

- Matches bean by type.
- It looks up in the configuration file for the class type of the property
- The names of the property and bean can be different in this case
- It invokes the setter method internally for autowiring.


XML Example:

```xml
<bean id="state" class="sample.State">
    <property name="name" value="UP"/>
</bean>
<bean id="city" class="sample.City" autowire="byType"/>
``
```

4. CONSTRUCTOR:

- Injects dependencies via constructor.
- It works similar to the “byType” mode, but it looks for the class type of the constructor arguments.

XML Example:

```xml
<bean id="state" class="sample.State">
    <property name="name" value="UP"/>
</bean>
<bean id="city" class="sample.City" autowire="constructor"/>
```

City.java

```java
public class City {
    private State state;
    public City(State state) { this.state = state; }
}
```

5. AUTODETECT:

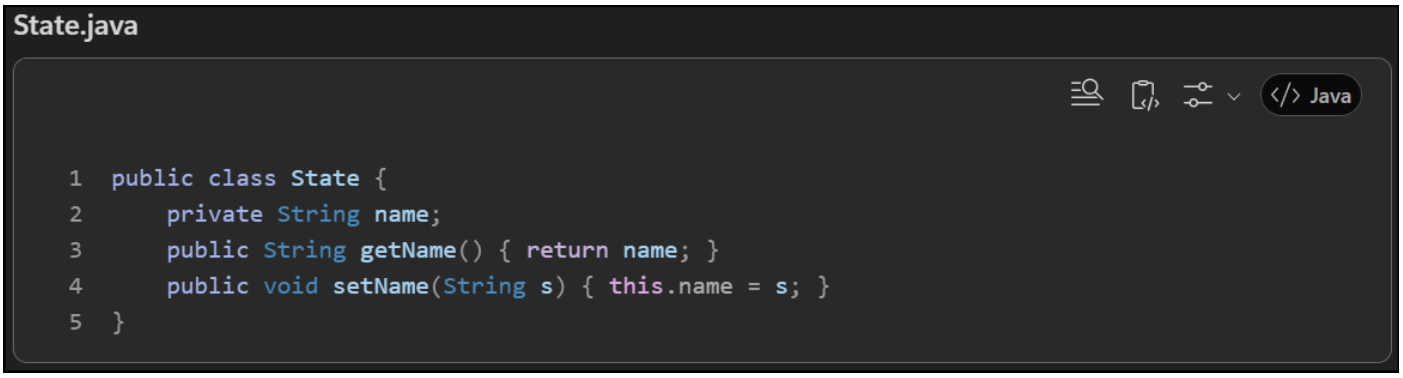
- Tries constructor first, then byType, deprecated since Spring 3.0.
- Instead of XML, use @Autowired
- No control for programmer and cannot inject primitives or Strings automatically.

Example:

```java
@Component
public class City {
    @Autowired
    private State state;
}
```

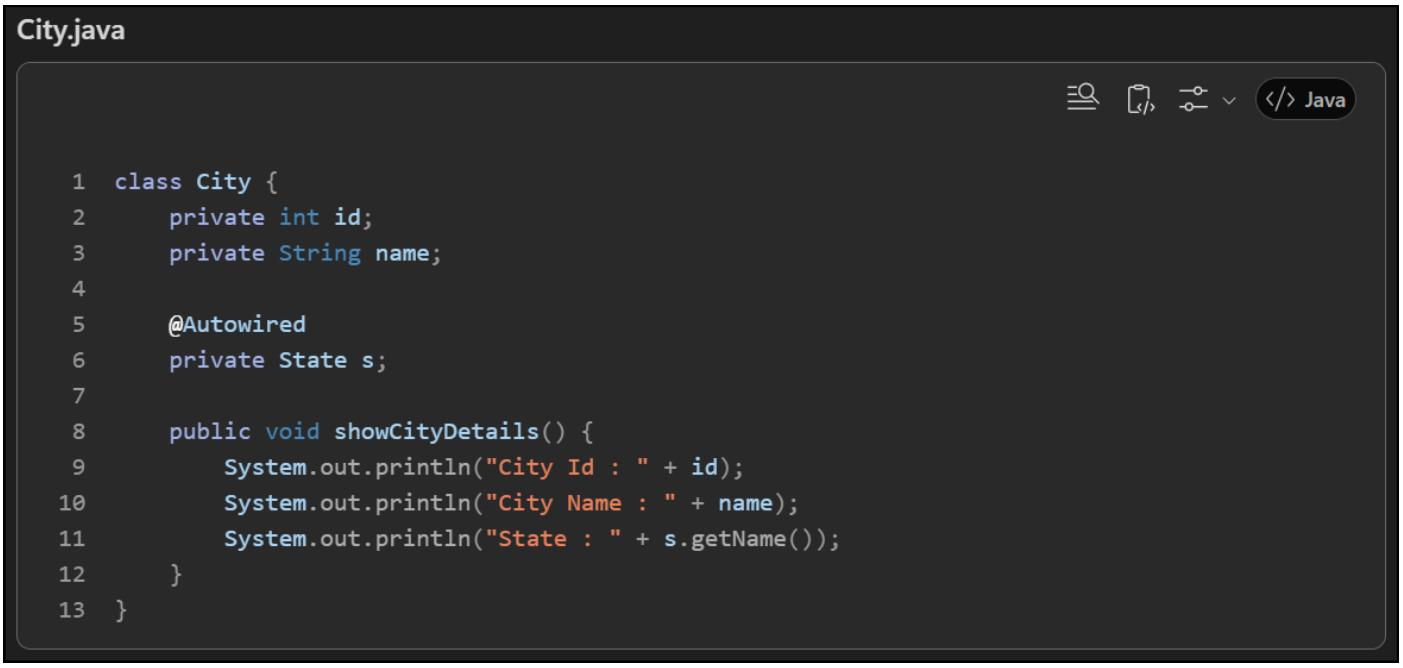
Or on constructor:

```java
@Component
public class City {
    private final State state;

    @Autowired
    public City(State state) {
        this.state = state;
    }
}
```



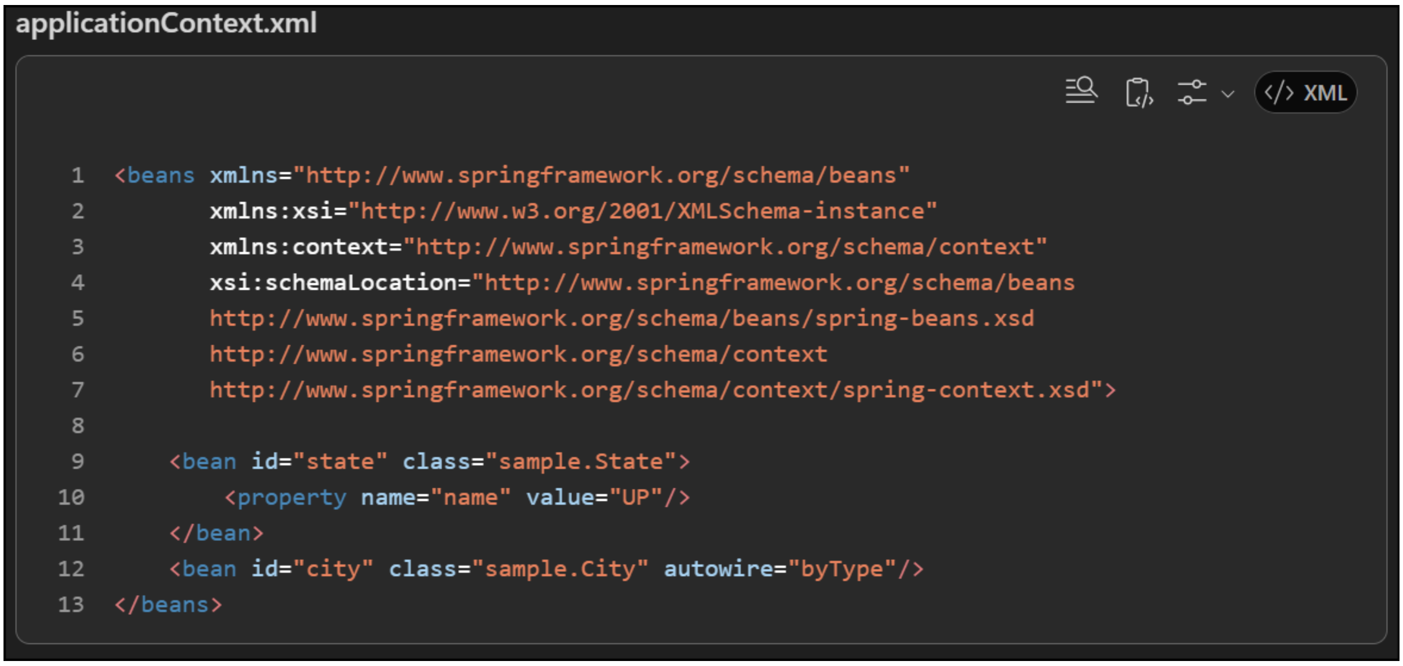
State.java

```java
public class State {
    private String name;
    public String getName() { return name; }
    public void setName(String s) { this.name = s; }
}
```



City.java

```java
class City {
    private int id;
    private String name;

    @Autowired
    private State s;

    public void showCityDetails() {
        System.out.println("City Id : " + id);
        System.out.println("City Name : " + name);
        System.out.println("State : " + s.getName());
    }
}
```



applicationContext.xml

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="state" class="sample.State">
        <property name="name" value="UP"/>
    </bean>
    <bean id="city" class="sample.City" autowire="byType"/>
</beans>
```

DISPATCHERSERVLET:

- **DEFINITION**: DispatcherServlet is the **Front Controller** in a Spring MVC application.
- **ROLE**: Acts as the **entry point** for all incoming HTTP requests.
- **FUNCTION**:
  - Receives the request.
  - Identifies the appropriate **controller** using handler mappings.
  - Delegates processing to the controller.
  - Returns the response to the client.

HOW IT WORKS:

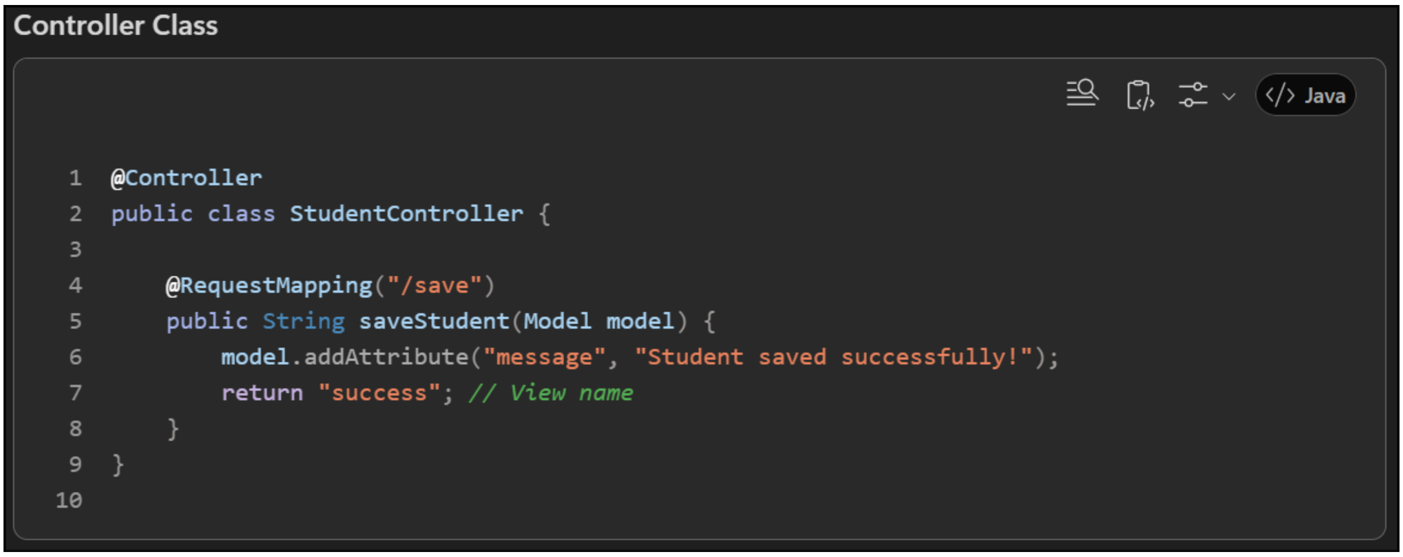
- **CLIENT REQUEST** → DispatcherServlet
- **DISPATCHERSERVLET**:
  - Uses **HANDLERMAPPING** to find the correct controller.
  - Delegates to **HANDLERADAPTER** to invoke the controller method.
- **CONTROLLER** processes the request and returns **MODEL AND VIEW**.
- **VIEWRESOLVER** resolves the view name to an actual view (e.g., JSP, **THYMELEAF**).
- **DISPATCHERSERVLET** renders the view and sends the response back.

KEY COMPONENTS:

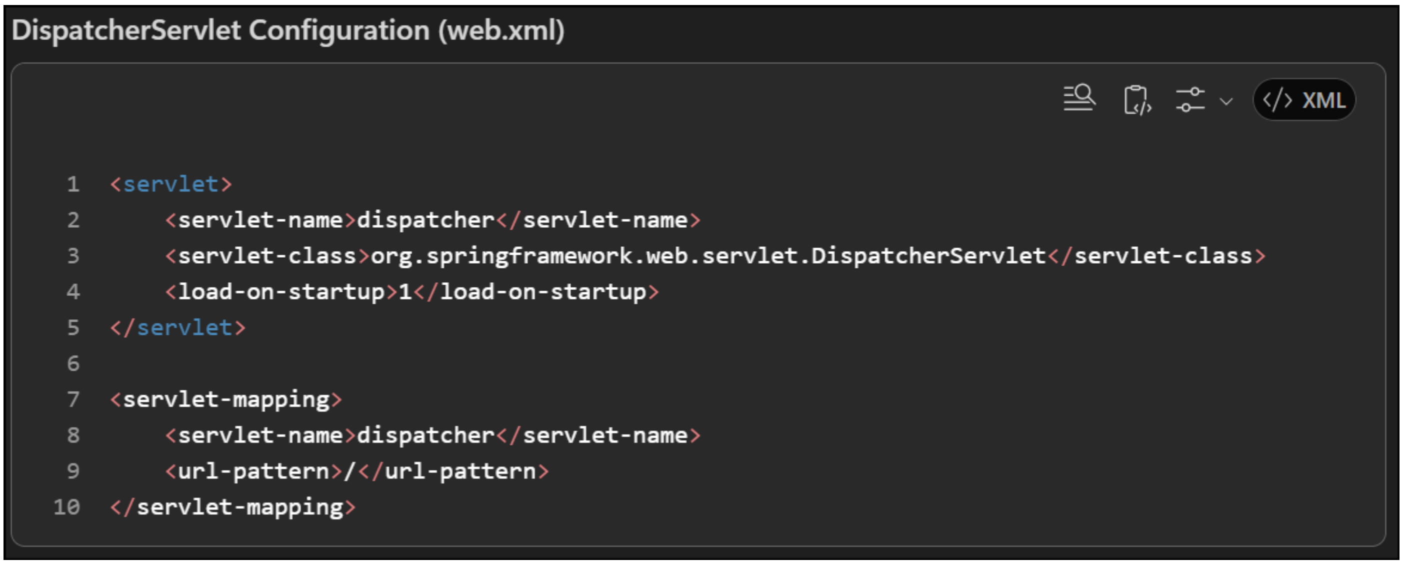
- **HANDLERMAPPING**: Maps URLs to controllers.
- **HANDLERADAPTER**: Invokes the controller method.
- **VIEWRESOLVER**: Resolves view names to actual views.
- **MODELANDVIEW**: Holds model data and view name.

REQUEST FLOW DIAGRAM:

CLIENT → DISPATCHERSERVLET → HANDLERMAPPING → CONTROLLER → VIEWRESOLVER → RESPONSE

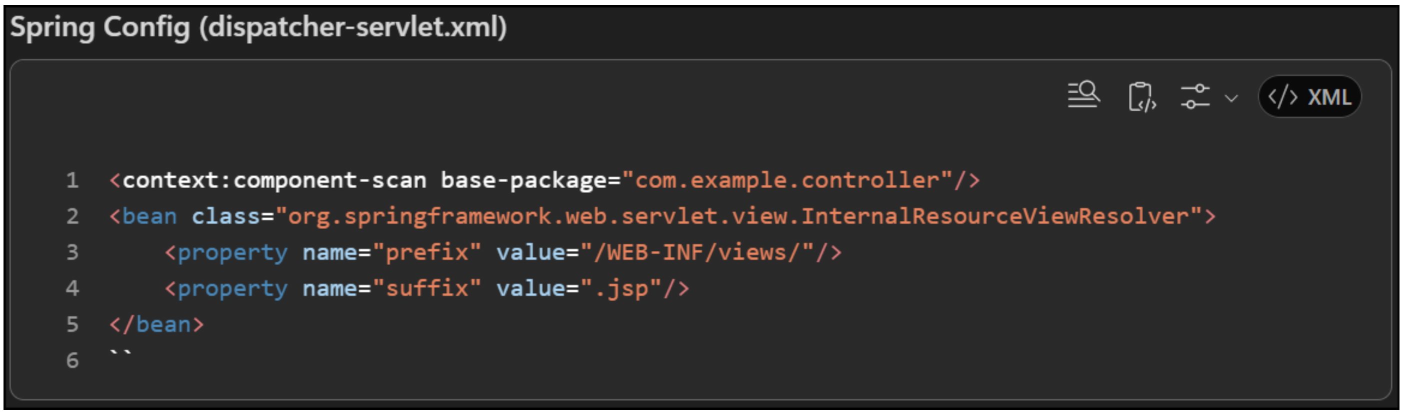

Controller Class

```java
@Controller
public class StudentController {

    @RequestMapping("/save")
    public String saveStudent(Model model) {
        model.addAttribute("message", "Student saved successfully!");
        return "success"; // View name
    }
}

```



DispatcherServlet Configuration (web.xml)

```xml
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```



Spring Config (dispatcher-servlet.xml)

```xml
<context:component-scan base-package="com.example.controller"/>
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".jsp"/>
</bean>
``
```

INTRODUCTION TO BUILD TOOLS:

Build tools are programs that **automate the steps required to convert source code into a deployable application**. These steps include:

- **COMPILING THE CODE**
- **RUNNING TESTS**
- **PACKAGING THE SOFTWARE**
- **DEPLOYING TO PRODUCTION**

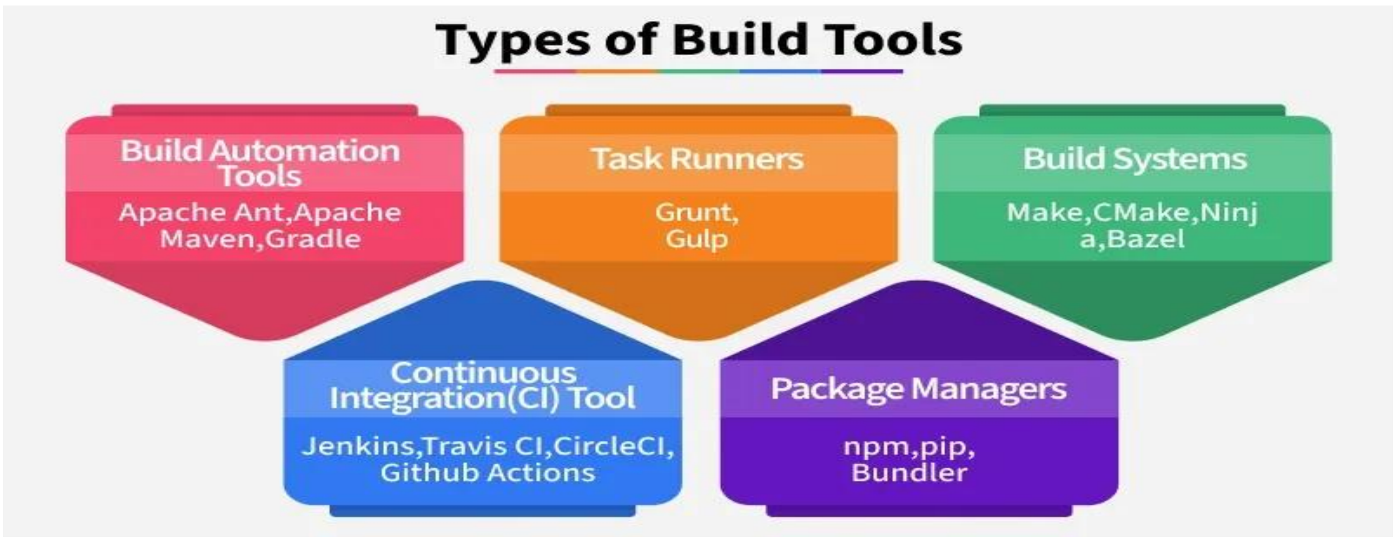
In **DevOps**, build tools ensure smooth, automated, and consistent integration and delivery whenever developers make changes.

WHY ARE BUILD TOOLS IMPORTANT:

Imagine you are building a Java project. Without a build tool, you must manually compile every .java file, run each test one by one, then bundle everything into a .jar file every time you make a change.

- **TIME-SAVING**
  - Automates repetitive tasks like compiling and packaging.
  - Example: mvn package in Maven compiles, tests, and packages in one command.
- **CONSISTENCY**
  - Ensures identical builds across different environments.
  - Example: Gradle guarantees same dependency versions on Windows, Mac, Linux.
- **ERROR REDUCTION**
  - Reduces human error by following predefined scripts.
  - Example: Ant uses build.xml to include all required files automatically.
- **SUPPORTS CI/CD**
  - Integrates with tools like Jenkins, GitHub Actions for automated pipelines.
  - Example: GitHub Actions triggers Maven or npm for build and test steps.
- **DEPENDENCY MANAGEMENT**
  - Automatically downloads and manages libraries.
  - Example: Maven fetches JUnit by adding it to pom.xml.

TYPES OF BUILD TOOLS:



1. BUILD AUTOMATION TOOLS:

Automate converting source code into executable programs.

- **Apache Ant**: Uses XML files to define build processes. It is flexible but can be verbose.
- **Apache Maven**: Introduces a standardized project structure and manages dependencies via a pom.xml file.
- **Gradle**: Combines the best features of Ant and Maven, using a Groovy or Kotlin DSL for configuration. It is known for its performance and flexibility.

2. TASK RUNNERS:

Primarily used in front-end development to automate repetitive tasks like minification, compilation, and testing.

- **Grunt**: A JavaScript task runner that uses a configuration file (Gruntfile.js) to define tasks.
- **Gulp**: Streams files through a series of plugins, allowing for faster builds compared to Grunt.

3. BUILD SYSTEMS:

Designed for large-scale projects, these systems focus on speed and scalability.

- **Make**: One of the earliest build tools, using Makefile to define build rules.
- **CMake**: Generates platform-specific build files, often used in C/C++ projects.
- **Ninja**: Emphasizes speed, making it suitable for projects with numerous small files.
- **Bazel**: Developed by Google, it handles builds and tests across multiple languages and platforms.

4. CONTINUOUS INTEGRATION (CI) TOOLS:

Automate the process of integrating code changes, running tests, and deploying applications.

- **Jenkins**: An open-source automation server that supports building, deploying, and automating software projects.
- **Travis CI**, **CircleCI**, **GitHub Actions**: Cloud-based CI services that integrate with version control systems to automate testing and deployment.

5. PACKAGE MANAGERS:

Manage project dependencies and can include basic build capabilities.

- **npm**: The default package manager for Node.js, handling JavaScript dependencies.
- **pip**: Python's package installer, managing libraries and dependencies.
- **Bundler**: Manages Ruby project dependencies.

TOP 5 BUILD TOOLS:

1. MAVEN

Maven is a popular build tool, especially for Java-based projects. It automates the build process and simplifies the management of project dependencies.

FEATURES:

- **Dependency Management:** Handles and updates libraries automatically.
- **Build Automation:** Uses a pom.xml file to define tasks and dependencies.
- **Repository Management:** Has a central place for storing and sharing libraries.
- **Plugin Support:** Offers plugins to automate tasks like testing and compiling.

ADVANTAGES:

- **Standardized Structure:** Makes Java projects have a consistent structure.
- **Cross-platform:** Works on different operating systems.
- **Large Ecosystem:** Plenty of plugins and integrations available.

LIMITATIONS:

- **Complexity:** Can be tricky for beginners due to its detailed XML configuration.
- **Slower Builds**: Larger projects may take more time to build.

2. GRADLE

Gradle is a flexible and fast build tool that works well for Java and many other languages. It's known for speeding up builds by only rebuilding parts of the project that have changed.

FEATURES:

- **Groovy-based:** Uses Groovy language for writing build scripts, making it easier to read.
- **Incremental Builds:** Rebuilds only changed parts of the project.
- **Multi-language Support:** Works with Java, C++, Python, and more.

ADVANTAGES:

- **Fast:** Gradle's incremental builds make it faster, especially for large projects.
- **Flexible:** Highly customizable to suit different needs.
- **Scalable**: Works well for both small and large projects.

LIMITATIONS:

- **Learning Curve:** It can take time to get used to Gradle's flexibility.
- **Less Standardized:** Its flexible setup can sometimes cause inconsistency.

3. ANT

Ant is an older, but highly customizable build tool. It allows you to define tasks like compiling or deploying code using XML files, giving you full control over the build process.

FEATURES:

- **XML-based Configuration:** Uses build.xml files to define tasks.
- **Task-based Build System:** Lets you specify and run tasks in a sequence.
- **Platform Independence**: Can be used across all platforms as it's written in Java.

ADVANTAGES:

- **Highly Customizable:** Gives you fine control over your build tasks.
- **Easy to Extend:** You can add custom tasks easily.

LIMITATIONS:

- **No Dependency Management:** Doesn't automatically handle external libraries, unlike Maven.
- **Verbose Configuration:** Requires a lot of setups, making it hard to maintain for larger projects.

4. MAKE

Make is a simple and lightweight build tool mostly used for C/C++ projects. It tracks file changes and rebuilds only what's needed, making it quick and efficient.

FEATURES:

- **Makefile System:** Defines build rules in a Makefile.
- **File Dependency Management:** Rebuilds only changed files.

ADVANTAGES:

- **Lightweight:** It's fast and simple to use.
- **Widely Supported:** Commonly used in Unix-like systems.

LIMITATIONS:

- **Limited to C/C++:** Primarily supports C/C++ projects and doesn't have support for other languages.
- **Manual Dependency Management:** Doesn't automatically handle dependencies like Maven.

5. BAZEL

Bazel is a fast, scalable build tool developed by Google. It is designed for building large, multi-language projects with a strong focus on performance and reproducibility.

FEATURES:

- **Multi-language support**: Works with Java, C++, Python, Go, and more.
- **High performance**: Uses advanced caching and parallelism for fast builds.
- **Scalable**: Ideal for large monorepos and complex codebases.
- **Hermetic builds**: Ensures builds are reproducible and isolated from the environment.

ADVANTAGES:

- **Speed**: Bazel rebuilds only what has changed, similar to Gradle, but optimized for massive codebases.
- **Cross-platform**: Works on Linux, macOS, and Windows.
- **Reproducibility**: Guarantees consistent builds regardless of environment.

LIMITATIONS:

- **Learning curve**: Configuration using Bazel's BUILD files can be unfamiliar for new users.
- **Complex setup**: Not as beginner friendly as Maven or npm.
- **Smaller ecosystem**: Compared to Maven or Gradle, fewer plugins and third-party tutorials.

CHOOSING THE RIGHT BUILD TOOL:

- **Project Size & Complexity**
  - Large projects → **Maven** or **Gradle** (dependency management, complex tasks).
  - Small/simple projects → **Ant** or **Make**.
- **Technology Stack**
  - **Java** → Maven, Gradle, Ant.
  - **C/C++** → Make, CMake.
  - **Multi-language** → Bazel.
- **Team Experience**
  - Maven and Ant → Easier for beginners (well-documented).
  - Gradle and Bazel → Require more expertise.
- **Build Speed**
  - Gradle and Make → Faster builds.
  - Maven → Slower for large projects.
- **CI/CD Integration**
  - Maven and Gradle → Strong integration with Jenkins, GitHub Actions, etc.